

# Streaming Virtual Machine to Stanford Streaming Architecture (part 2)

---

Abhishek Das

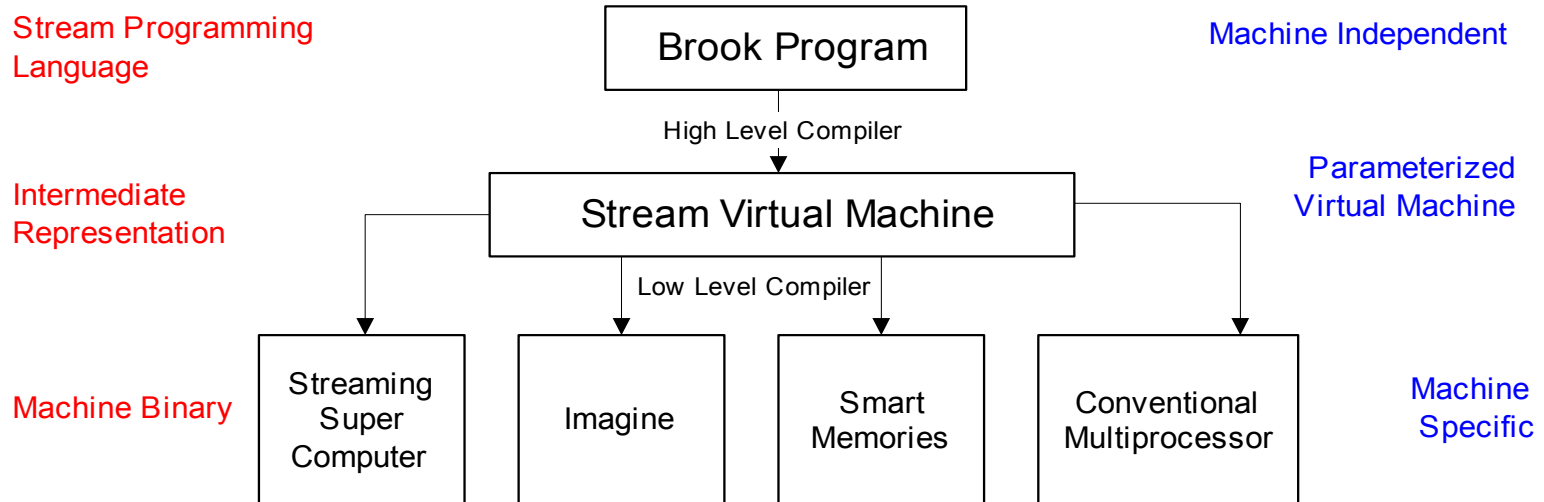
Mattan Erez

3/9/2004

# Streaming Virtual Machine(SVM)

---

- A Virtual Machine is a description of a specific architecture using a common set of architectural primitives
- SVM abstracts and parameterize features of different stream architectures
- Intermediate target for high level compiler to do resource allocation for various stream architectures



# SVM APIs

---

- Stream Virtual Machine API: set of constructs to express streaming portion of any application
- Goals:
  1. Express high performance mappings for multiple applications and architectures
  2. Separate computation- and data-intensive code from control code
  3. Explicitly manage memory for data-intensive code
    - Use DMA instead of caches
  4. Represent streaming nature of application
    - Explore hardware support for streaming
  5. Easy to translate by low-level compiler
  6. Human comprehensible and writable

# SVM API Constructs

---

- Kernel (Slave)
  - Computation- and data- intensive function (consumes and produces streams) executed on a single stream processor
  - Execution state
  - May not correspond to a “conceptual” kernel e.g. pre-defined kernels for data movement
  - Subset of C and SVMs
- Stream and Block
  - Data stream/block operated on by kernel, bound to specific resource
  - Streams: push, pop, peek, EOS
- Control (Master)
  - Scalar control code
  - Initiate, monitor and terminate kernels
  - Subset of C and SVM APIs

# Kernels

---

- Status:



- Kernel structure:

- input/output streams/blocks
  - other fields accessible to control thread

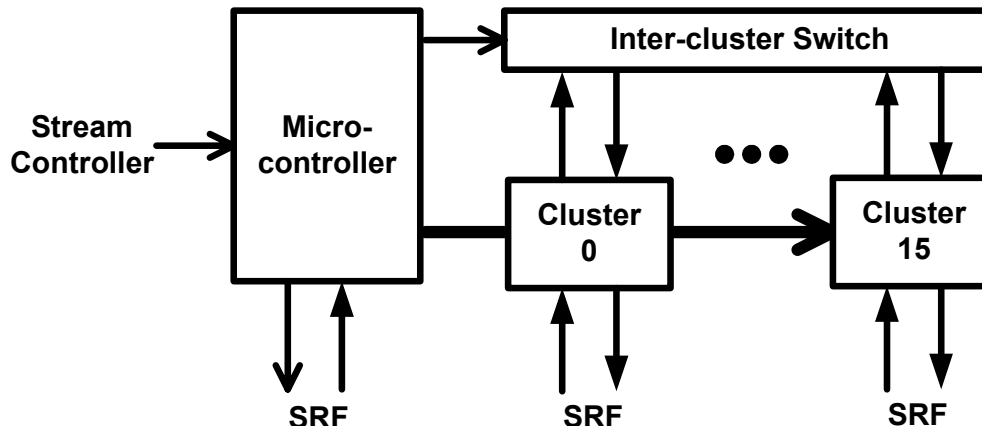
- Control methods:

- KernelInit(Kernel \*,...): initialization
    - Processor resource
    - Spilling memory (optional)
  - KernelWork(Kernel \*k): computation
  - addDependence(Kernel \*dependsOnKernel)
  - KernelReady(), KernelWait(), KernelRun(), KernelPause(), KernelEnd()

# Compilation target

---

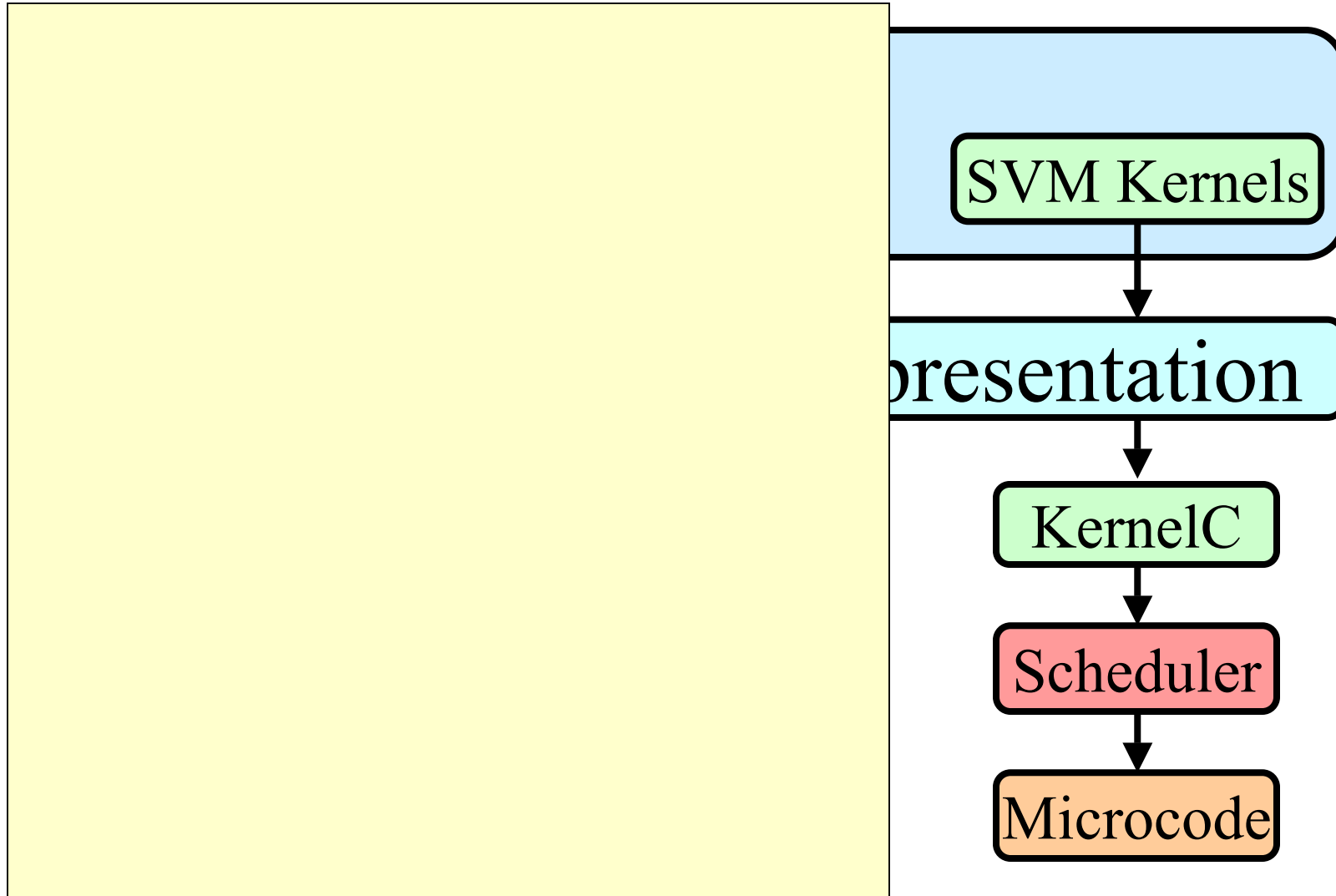
- High-level compiler
  - Extracts/merges/splits kernels, streams, blocks
  - Maps kernels to processors
  - Maps streams and blocks to memory
  - Performs global dependency analysis and inserts `addDependency()` and `wait()` as needed
- Low-level compiler
  - Translates SVM constructs to hardware
  - **Compiles kernels**



Target:  
Stanford  
Streaming  
Architecture

# The whole story

---



# SVM APIs

---

- `typedef struct { } Stream;`
- `typedef Stream (I)(O)Stream;`
  
- `typedef struct { } Kernel;`
  
- `void streamPush(Stream* s, void* element)`
- `void streamPushWithEOS(Stream* s, void* element)`
- `void streamPop(Stream* s, void* element)`
- `bool streamPeekEOS(Stream* s, int n)`
- `void streamPeek(Stream* s, int n, void* element)`

# SVM Kernel code restrictions (C)

---

1. No pointers except for SVM constructs.
2. No dynamic memory allocation.
3. No accesses to global variables.
4. Support for arrays with a fixed (int literal) length and struct's containing members of any other type.
5. No GOTO statements (all control flow is structured).
6. No recursive functions (all function calls have inline semantics).
7. Supported opcodes are only the logical, arithmetic, and boolean operations found in C (use math library).

# KernelC

---

1. Basic types: (u)int, float, packed data types
  - bool: cc (condition code)
  - struct: user defined records
2. (c)istream<>, (c)ostream<>, uc<>: visible outside clusters
3. array<>, expand<>: fixed (int literal) length
4. No pointers and no global memory access.
5. Control flow:
  - loop\_stream (stream), loop\_count (uc)
  - loop\_while\_any (cc), loop\_while\_all (cc)
  - loop\_until\_any (cc), loop\_until\_all (cc)
6. No recursive functions (all function calls have inline semantics).
7. Logical, Arithmetic and communication opcodes.<sup>10</sup>

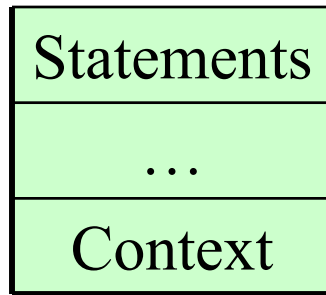
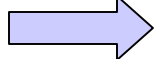
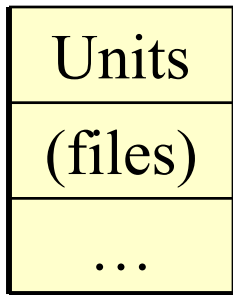
# Ctool

---

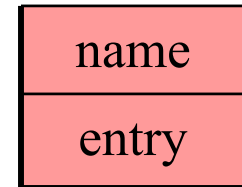
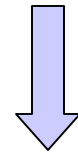
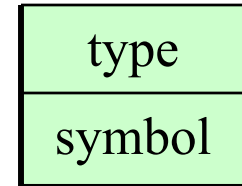
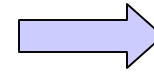
- Aim: Create a parsing library that could be used as the basis of source-transforming tools.
- CTool is a C lexer/parser with a symbol table.
- Generates *abstract syntax tree* - so you can easily parse C code, make modifications to the AST, and reprint it.
- License: GNU Library or Lesser General Public License (LGPL)
- Programming Language: C++

# Symbol Table and AST

Project

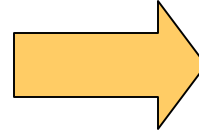
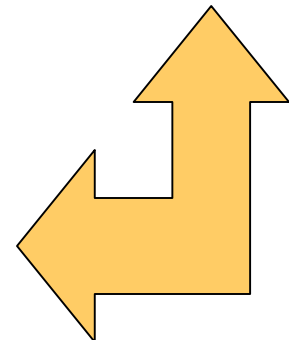


- Declaration
- Expression
- If
- For
- While
- DoWhile
- Block

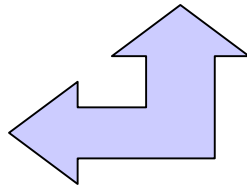
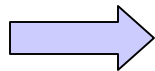
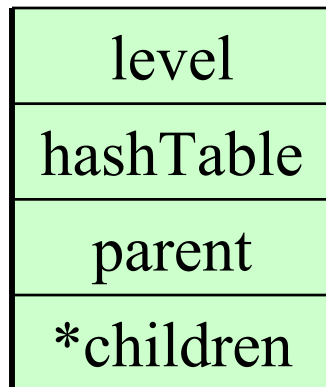


SymEntry

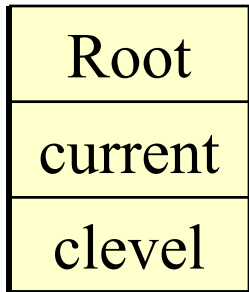
- name
- symType
- decl
- def
- scope



Scope Table



Symbol Table



# Example Kernel

---

```
typedef struct {
    Kernel kernel;
    IOStream* io;
    int mul;
} MulKernel;

void mulKernelWork(MulKernel* k) {
    int x;

    for (x = 0; x < k->mul; x++) {
        streamPush(k->io, &x);
    }

    while (!streamPeekEOS(k->io, 0)) {
        streamPop(k->io, &x);
        k->mul *= x;
    }
}
```

```
kernel mulKernel (
    iostream<int> io,
    uc<int>& _UC_mul) {
    int x;
}
```

# Example Kernel

---

```
typedef struct {
    Kernel kernel;
    IOStream* io;
    int mul;
} MulKernel;
```

```
void mulKernelWork(MulKernel* k) {
    int x;
```

```
    for (x = 0; x < k->mul; x++) {
        streamPush(k->io, &x);
    }
```

```
    while (!streamPeekEOS(k->io, 0))
    {
        streamPop(k->io, &x);
        k->mul *= x;
    }
}
```

```
kernel mulKernel(iostream<int> io,
                 uc<int>& _UC_mul) {
    int x;
    int _CL_mul;
    uc<int> _UCount;

    _CL_mul= commclperm(ucid(), x, _UC_mul);
    _CL_mul = idiv_iter(_CL_mul,NUM_CLUSTERS);
    _CL_mul= commclperm(0,_CL_mul,_UCount);

    loop_count(_UC_mul_count) {
        io << x;
    }
}
```

# Example Kernel

```
typedef struct {  
    Kernel kernel;  
    IOStream* io;  
    int mul;  
} MulKernel;
```

```
void mulKernelWork(MulKernel* k) {  
    int x;  
  
    for (x = 0; x < k->mul; x++) {  
        streamPush(k->io, &x);  
    }
```

```
    while (!streamPeekEOS(k->io, 0)) {  
        streamPop(k->io, &x);  
        k->mul *= x;  
    }
```

```
}
```

```
kernel sumKernel(iostream<int> io,  
                uc<int>& _UC_mul) {  
    int x;  
    int _CL_mul;  
  
    loop_stream(io) {  
        io >> x;  
  
        _CL_mul = commclperm(ucid(),x,_UC_mul);  
  
        _CL_mul = _CL_mul * x;  
        _CL_mul = commclperm(0, _CL_mul);  
  
        ...  
        [NUM_CLUSTERS ]  
  
        _CL_mul = commclperm(NUM_CLUSTERS-1,  
                            _CL_mul, NUM_CLUSTERS-1, mul);  
    }  
}
```

# Kernel conversion

---

## 1. Structures and types:

- Kernel and stream symbol table entries
- Kernel fields : *uc* variables
- structs : *records*
- Stream types derived from stream I/O

## 2. Expressions - Arithmetic operations:

- $\text{sqrt}$  : *fsqrt\_iter*
- $\text{fabs}$  : *trunc*
- $x \% y$  : *imod\_iter(x, y)*
- $x / y$  : *idiv\_iter(x, y), fdiv\_iter(x, y)*
- all C math library operations (may) not (be) supported
- **Optimizations:** *finvsqrt\_iter (1/sqrt(x)), finv\_iter (1/x)*

## 3. Expressions - boolean operations:

- Types changed to *cc*

# Kernel conversion

---

## 4. Control flow :

- for loop:
  1.  $x = \text{select}(\text{itocc}(\text{cid}() == k), \text{start} + (k * \text{increment}), \text{start});$  //loop preamble
  2.  $x = x + \text{increment};$  // end of loop block
- *uc* operation: increment/decrement by 1 for kernel fields
- loop condition :
  - *uc* : *loop\_count*
  - *stream* : *loop\_stream*
  - *cond* ( $x < \text{end}$ ) :
    - *loop\_while\_all* (*itocc* (*cond*) )
    - *if-else* on *cid*()
- if-else : **Predication / Conditional streams: Ujval's thesis**

## 5. Reduction : inter-cluster communication

- **Optimize for commutative and associative operations**
- **Loop nesting**

# Kernel conversion

---

## 6. Arrays :

- Scratchpad : extract from SVM kernel initialization
- expand : in LRF
  - multi-dimension – Region based analysis / Induction variables
- Introduce new indexable streams

## 7. Function calls : force *inline* functions (SVM assures semantics)

## 8. Stream operators (Function calls – symbol table entries):

- streamPush : <<
- streamPop : <<
- streamPeekEOS(0) : *loop\_stream*
- streamPushWithEOS :: conditional output stream:
  - *costr(cond) << x;*
  - set *cond* false and end with *flush(costr, EOS)*
- **streamPeek(int n)** and **streamPeekEOS(int n)**

# Kernel conversion

---

- `streamPeekEOS(int n)` :
  - read stream length during kernel startup
  - keep track of number of words read
- `streamPeek(int n)` :
  - Inter-cluster communication
    - ugly
    - only  $n < 0$
    - `streamPeek(str, -1, &y)` : (Assume x has current value)
      - » `uc<type> _UC_oldx = 0; // initialization`
      - » `int perm = select( itocc(cid()) == 0), ucid(), cid()-1);`
      - » `y = commclperm (perm, x, _UC_oldx, NUM_CLUSTERS-1, _UC_oldx);`

# Kernel conversion

---

- `streamPeek(int n)` and `streamPeekEOS(int n)` :
  - Index streams for flexibility in  $n$ 
    - Stream marker for `streamPeek(str, n, &y)` :
      - » `marker<type> m = &str[n]; // initialization`
      - » `m >> y;`
      - » `m += NUM_CLUSTERS;`
    - keep track of number of words read: now need one more marker for `streamPop`
    - optimize using region based analysis / Induction variables
    - peeking beyond bounds : conditional index streams
  - Conditional Index streams
    - Stream marker for `streamPeekEOS(str, n)` :
      - » `marker<type> m = &str[pos]; // initialization`
      - » `m(ccend) >> y;`
      - » `m += NUM_CLUSTERS;`
    - *ccend* is set if no elements available

# Kernel conversion

---

- `streamPeek(int n)` and `streamPeekEOS(int n)` :
  - Index streams can become a big overhead
  - If  $n$  is constant and the rate of *peek* and *pop* is static, *markers* are only required for initialization
    - Need techniques to handle stream start address
  - Copy propagation is not trivial: requires inter-cluster communication

# Conclusion and Future work

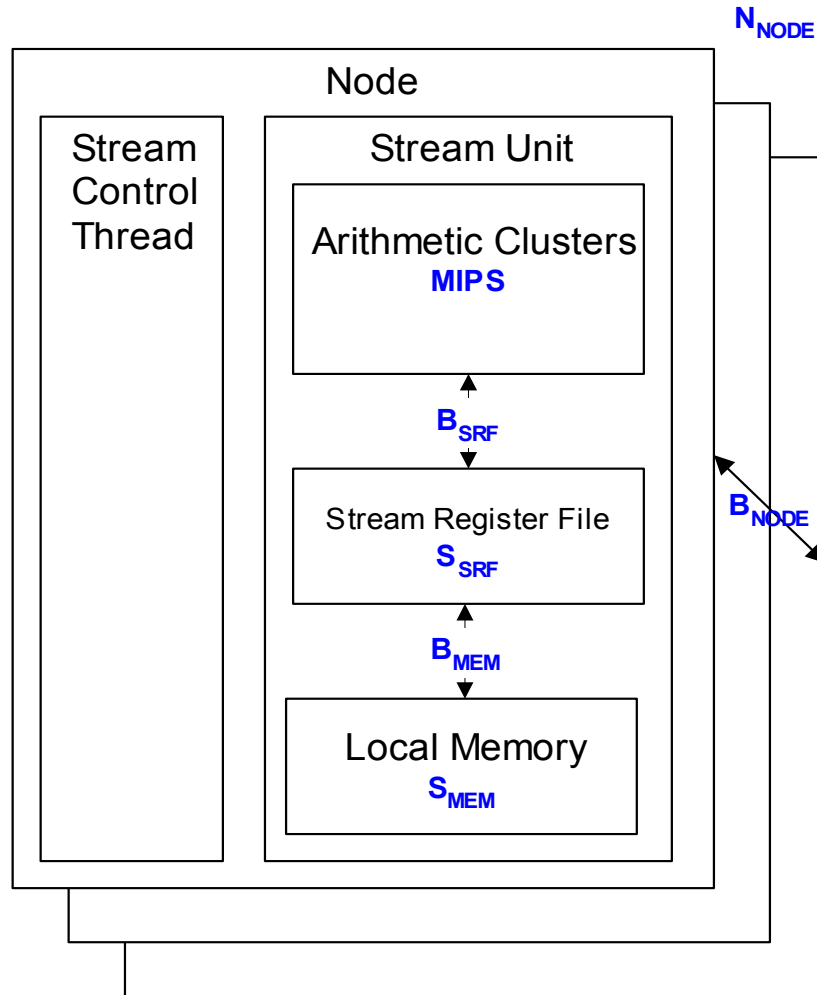
---

- Design for running SVM kernels on stanford streaming architectures
- **Finished!** first cut implementation
- Resource allocation : kernel restructuring

# Backup slides

# SVM Parameters

---

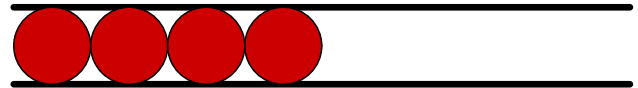


# Streams and Blocks

---

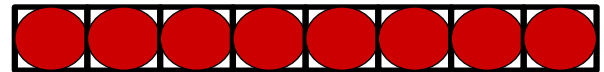
- Stream is a queue of records(sequential data elements) buffered in memory or registers:

- Push, pop, and peek
- EOS tags
- Capacity, length, totalLength



- A Block is an array of records(indexed) in a specific memory or registers:

- Read, write
- Capacity

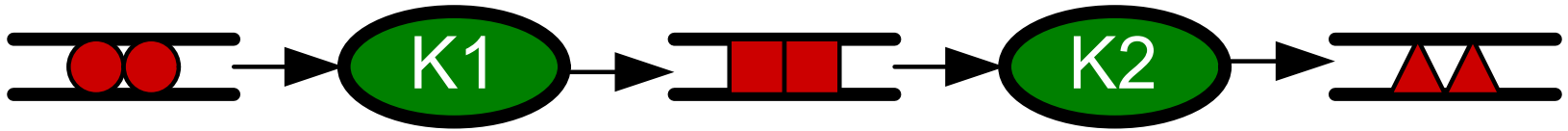


- Initialization:

- streamInitRAM()
- streamInitWithDataRAM()
- streamInitFIFO()

# SVM Mapping Example

- Application:



- Architecture and Mapping:

